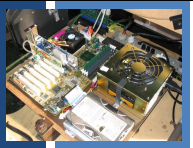
TCSS 422: OPERATING SYSTEMS



Three Easy Pieces: Translation Lookaside Buffer, Paging – Smaller Tables

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Assignment 3 – Page Table Walker
- Review Quiz #5 –Memory Virtualization
- **Memory Virtualization**
- Wrap-up: Translation Lookaside Buffer – Ch. 19
- Paging – Smaller Tables – Ch. 20
- Beyond Physical Memory – Ch. 21/22

FEEDBACK – 5/16

- How to determine whether a cache lookup is a hit or a miss?
- For example: for a TLB lookup?
- How do you get a miss or a hit in a page table?

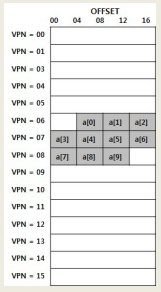
CHAPTER 19: TRANSLATION LOOKASIDE BUFFER (TLB)

OBJECTIVES

- Chapter 19
  - TLB Algorithm
  - TLB Tradeoffs
  - TLB Context Switch

TLB EXAMPLE - 3

```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```



- For the accesses: a[0], a[1], a[2], a[3], a[4],
- a[5], a[6], a[7], a[8], a[9]
- How many are hits?
- How many are misses?
- What is the hit rate? (%)
  - 70% (3 misses one for each VP, 7 hits)

EXAMPLE: LARGE ARRAY ACCESS

- Example: Consider an array of a custom struct where each struct is 64-bytes. Consider sequential access for an array of 8,192 elements stored contiguously in memory:
- 64 structs per 4KB page
- 128 total pages
- TLB caches stores a maximum of 64 - 4KB page lookups
- How many hits vs. misses for sequential array iteration?
  - 1 miss for every 64 array accesses, 63 hits
  - Complete traversal: 128 total misses, 8,064 hits (98.4% hit ratio)

TLB EXAMPLE IMPLEMENTATIONS

- Intel Nehalem microarchitecture 2008 – multi level TLBs
  - First level TLB: separate cache for data (DTLB) and code (ITLB)
  - Second level TLB: shared TLB (STLB) for data and code
  - Multiple page sizes (4KB, 2MB)
  - Page Size Extension (PSE) CPU flag for larger page sizes

Cache		Page Size	
Name	Level	4 KB	2 MB
DTLB	1st	64	32
ITLB	1st	128	7 / logical core
STLB	2nd	512	none

- Intel Haswell microarchitecture 22nm 2013
  - Two level TLB
  - Three page sizes (4KB, 2MB, 1GB)
- Without large page sizes consider the # of TLB entries to address 1.9 MB of memory…

Cache		Page size		
Name	Level	4 KB	2 MB	1 GB
DTLB	1st	64	32	4
ITLB	1st	128	8 / logical core	none
STLB	2nd	1024		none

HW CACHE TRADEOFF

- **Speed vs. size**

Speed ⟷ Size

- In order to be fast, caches must be small
- Too large of a cache will mimic physical memory
- Limitations for on chip memory

Dwight on "tradeoffs"

HANDLING TLB MISS

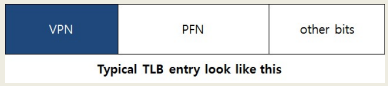
- Historical view
- CISC – Complex instruction set computer
  - Intel x86 CPUs
  - Traditionally have provided on CPU HW instructions and handling of TLB misses
  - HW has a page table register to store location of page table

HANDLING TLB MISS - 2

- RISC – Reduced instruction set computer
  - ARM CPUs
  - Traditionally the OS handles TLB misses
  - HW raises an exception
  - Trap handler is executed to handle the miss
- Advantages
  - HW Simplicity: simply needs to raise an exception
  - Flexibility: OS provided page table implementations can use different data structures, etc.

TLB CONTENTS

- TLB typically may have 32, 64, or 128 entries
- HW searches the entire TLB in parallel to find the translation
- Other bits
  - Valid bit: valid translation?
  - Protection bit: read/execute, read/write
  - Address-space identifier: identify entries by process
  - Dirty bit



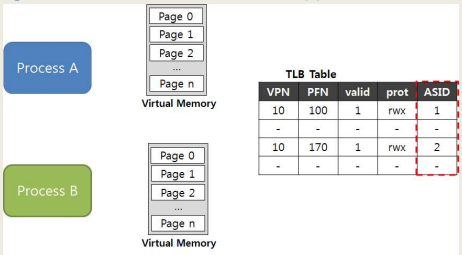
Typical TLB entry look like this

TLB: ON CONTEXT SWITCH

- TLB stores address translations for current running process
- A context/switch to a new process invalidates the TLB
- Must "switch" out the TLB
- **TLB flush**
  - Flush TLB on context switches, set all entries to 0
  - Requires time to flush
  - TLB must be reloaded for each C/S
  - If process not in CPU for long, the TLB may not get reloaded
- **Alternative**: be lazy…
  - Don't flush TLB on C/S
  - Share TLB across processes during C/S
  - Use address space identifier (ASID) to tag TLB entries by process

TLB: CONTEXT SWITCH - 2

- Address space identifier (ASID): enables TLB data to persist during context switches – also can support virtual machines



TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

SHARED MEMORY SPACE

VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

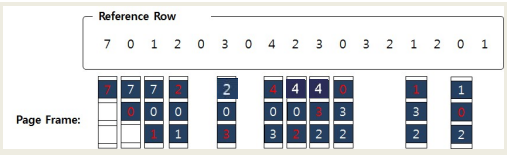
- When processes share a code page
  - Shared libraries ok
  - Code pages typically are RX, not RWX

Sharing of pages is useful as it reduces the number of physical pages in use.

CACHE REPLACEMENT POLICIES

- When TLB cache is full, how add a new address translation to the TLB?
- Observe how the TLB is loaded / unloaded…
- Goal minimize miss rate, increase hit rate
- **Least Recently Used (LRU)**
  - Evict the oldest entry
- **Random policy**
  - Pick a candidate at random to free-up space in the TLB

LEAST RECENTLY USED



- RED – miss
- WHITE – hit
- For 3-page TLB, observe replacement

11 TLB miss, 5 TLB hit

CHAPTER 20: PAGING: SMALLER TABLES

OBJECTIVES

- Chapter 20
  - Smaller tables
  - Hybrid tables
  - Multi-level page tables

LINEAR PAGE TABLES

- Consider array-based page tables:
  - Each process has its own page table
  - 32-bit process address space (up to 4GB)
  - With 4 KB pages
  - 20 bits for VPN
  - 12 bits for the page offset

LINEAR PAGE TABLES - 2

- Page tables stored in RAM
- Support potential storage of 2^{20} translations
  = 1,048,576 pages per process @ 4 bytes/page
- Page table size 4MB / process

Page table size = $\frac{2^{32}}{2^{12}} * 4Byte = 4MByte$

- Consider 100+ OS processes
  - Requires 400+ MB of RAM to store process information

LINEAR PAGE TABLES - 2

- Page tables stored in RAM
- Support potential storage of 2^{20} translations
  = 1,048,576 pages per process @ 4 bytes/page
- Page table size 4MB / process

Page tables are too big and consume too much memory.

Need Solutions …

- Consider 100+ OS processes
  - Requires 400+ MB of RAM to store process information

PAGING: USE LARGER PAGES

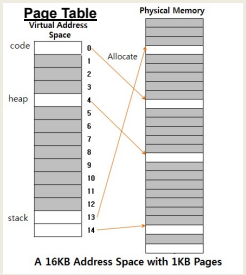
- Larger pages = 16KB = 2^{14}
- 32-bit address space: 2^{32}
- 2^{18} = 262,144 pages

$\frac{2^{32}}{2^{14}} * 4 = 1MB$ per page table

- Memory requirement cut to ¼
- However pages are huge
- Internal fragmentation results
- 16KB page(s) allocated for small programs with only a few variables

PAGE TABLES: WASTED SPACE

- Process: 16KB Address Space w/ 1KB pages



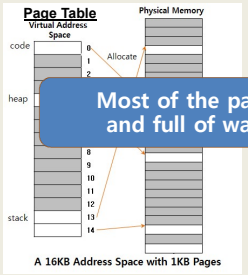
A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
…	…	…	…	…
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

PAGE TABLES: WASTED SPACE

- Process: 16KB Address Space w/ 1KB pages



A 16KB Address Space with 1KB Pages

Most of the page table is unused and full of wasted space. (73%)

PFN	valid	prot	present	dirty
15	1	rw-	1	1
...	...	...	...	...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

MULTI-LEVEL PAGE TABLES

- Consider a page table:
- 32-bit addressing, 4KB pages
- 2^{20} page table entries
- Even if memory is sparsely populated the *per process* page table requires:

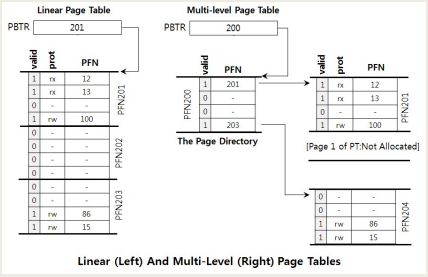
$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4Byte = 4MByte$$

- Often most of the 4MB *per process* page table is empty
- Page table must be placed in 4MB contiguous block of RAM

- MUST SAVE MEMORY!

MULTI-LEVEL PAGE TABLES - 2

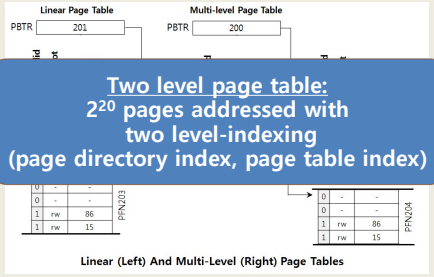
- Add level of indirection, the "page directory"



Linear (Left) And Multi-Level (Right) Page Tables

MULTI-LEVEL PAGE TABLES - 2

- Add level of indirection, the "page directory"



Two level page table:
2^{20} pages addressed with two level-indexing (page directory index, page table index)

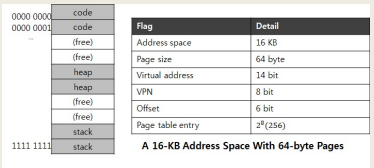
Linear (Left) And Multi-Level (Right) Page Tables

MULTI-LEVEL PAGE TABLES - 3

- Advantages
  - Only allocates page table space in proportion to the address space actually used
  - Can easily grab next free page to expand page table
- Disadvantages
  - Multi-level page tables are an example of a time-space tradeoff
  - Sacrifice address translation time (now 2-level) for space
  - Complexity: multi-level schemes are more complex

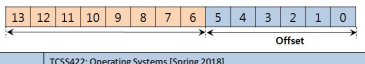
EXAMPLE

- 16KB address space, 64byte pages
- How large would a one-level page table need to be?
- 2^{14} (address space) / 2^{6} (page size) = 2^{8} = 256 (pages)



Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	2^{8}(256)

A 16-KB Address Space With 64-byte Pages

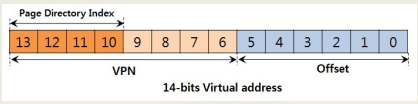


EXAMPLE - 2

- 256 total page table entries (64 bytes each)
- 1,024 bytes page table size, stored using 64-byte pages = (1024/64) = 16 page directory entries (PDEs)
- Each page directory entry (PDE) can hold 16 page table entries (PTEs) *e.g. lookups*
- 16 page directory entries (PDE) x 16 page table entries (PTE) = 256 total PTEs
- **Key Idea: the page table is stored using pages too!**

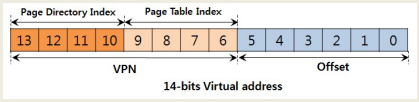
PAGE DIRECTORY INDEX

- Now, let's split the page table into two:
  - 8 bit VPN to map 256 pages
  - 4 bits for page directory index (PDI – 1st level page table)
  - 6 bits offset into 64-byte page



PAGE TABLE INDEX

- 4 bits page directory index (PDI – 1st level)
- 4 bits page table index (PTI – 2nd level)



- To dereference one 64-byte memory page,
  - We need one page directory entry (PDE)
  - One page table Index (PTI) – can address 16 pages

EXAMPLE - 3

- **For this example, how much space is required to store as a *single-level* page table with any number of PTEs?**
- 16KB address space, 64 byte pages
- 256 page frames, 4 byte page size
- 1,024 bytes required (*single level*)
- **How much space is required for a *two-level* page table with only 4 page table entries (PTEs) ?**
- Page directory = 16 entries x 4 bytes (1 x 64 byte page)
- Page table = 4 entries x 4 bytes (1 x 64 byte page)
- 128 bytes required (2 x 64 byte pages)
  - Savings = using just 12.5% the space !!!

QUESTIONS